Amendments to the Specification

Please amend the title of the invention on the cover page and on page 1 as follows: ~~SYSTEM AND METHOD FOR EXECUTING PREDICATED CODE OUT OF ORDER~~ SYSTEM AND METHOD FOR PREVENTING PREDICATED INSTRUCTIONS FROM DELAYING EXECUTION OF CONSUMER INSTRUCTIONS.

Please amend the first full paragraph on page 2 as follows:

One example of an advanced microarchitecture is that of a dynamic, or out-of-order, execution model. An out-of-order~~,~~ execution model ~~is, in general,~~ is generally more complex than a static execution model. Static execution executes code in the order as scheduled statically by the compiler ~~while out-of-order~~, whereas out-of-order execution permits the processor to dynamically adjust instruction scheduling to the run-time behavior of the program. Because of this ability to adapt to the run-time environment, dynamic execution has been employed in many processor designs. The potential performance gains of an ~~out-of-order~~ out-of-order execution model are facilitated by two techniques: ~~Register~~ i) register renaming where registers are renamed to eliminate false dependencies; and ii) ~~and~~ dynamic scheduling where instructions are reordered to reduce unnecessary stalls in the pipeline.

Please amend the paragraph bridging pages 5-6 as follows:

There are several types and variations of ~~an out-of-order~~ out-of-order or dynamic execution processors. A dynamic microarchitecture as a baseline performance embodiment is shown in Fig. 1. The baseline performance embodiment includes a dynamic portion 105 of the processor 100, including a register renaming unit 110~~, which~~

that maps between temporary and architectural files, a reorder buffer 120, a plurality of reservation stations 130, and a plurality of execution units 140. A bus 115 couples the register renaming unit 110, the reorder buffer 120, the plurality of reservation stations 130 and the plurality of execution units 140 together and to the remaining portions of the microprocessor which are not shown. The pipeline shown in Fig. 1A has 15 stages, with 7 stages 155-161 devoted to the dynamic portion 105 of the processor 100. The dynamic pipeline 155-161 begins with a 2-stage rename 155-156, followed by a register read stage 157, a 2-stage schedule 158-159, an execute stage 150, and finally a retire stage 161. In the schedule stage 158-159, the instructions wait in the reservation stations 130 until the data of the source operands become available. After the data from the source operands are loaded into the register, the instruction enters the execute stage 150. In the final retire stage 161, the instructions are retired in order from the reorder buffer.

Please amend the first full paragraph on page 8 as follows:

The performance of a dynamic execution processor can degrade with the above described predicated code sequence. When a consumer instruction reaches the rename stage 155, 156, the renaming of the common register becomes ambiguous if the guarding predicates of the defining instructions are not resolved. In the middle 190 of Fig. 1C, two add instructions, guarded by p9 and p3, assign their respective results to the same architectural register r40. After renaming 194, the result register is renamed to rB and rC, respectively. A mov instruction that uses or consumes the result register follows immediately in the pipeline. If the mov instruction enters the rename stage before predicates p9 and p3 are evaluated, then the processor cannot correctly

Q3 Cntd.

determine whether to rename r40 to physical registers rB or rC. Therefore, the processor stalls the ~~consumer instruction, the~~ mov instruction ~~before entering the mov instruction~~ before entering it into the rename stage.

Please amend the first full paragraph on page 9 as follows:

A consumer instruction is not required to wait for the resolution of all guarding predicates of the defining instructions as shown in Fig. 2A. The consumer instruction must only wait for the latest defining instruction that is guarded true. Therefore, the consumer instruction first waits for the predicate of the last of the defining instructions to become available 256. If the predicate of the last of the defining instructions turns out true 258, the consumer instruction can immediately advance in the pipeline 200 and, in this example, use the physical register of the last defining instruction, despite the outcome of other defining instructions. If the last defining instruction is not true ~~i.e. nullified~~ (i.e., it is nullified), then the consumer instruction must wait for the predicate of the second-to-last defining instruction 260. The process repeats until a latest defining instruction is guarded true. This prioritized checking scheme for the predicate values affects performance depending on the order those values become available. It will be further appreciated that the instructions represented by the blocks in Fig. 2A ~~is~~ are not required to be performed in the order illustrated, and that all the processing represented by the blocks may not be necessary to practice the invention.

Q4

Please amend the paragraph bridging pages 9 and 10 as follows:

Q5

According to baseline performance embodiment described above, the simple dynamic processor that runs predicated code could suffer from excessive pipeline stalls

Q5 Cont'd.

due to scheduling and renaming issues ~~as described above~~. One alternative embodiment postpones the predicated instructions down the pipeline and resolves the predicated instructions without significant change to the existing dynamic execution microarchitecture.

Please amend the first full paragraph on page 10 as follows:

Q6

For one embodiment, a select-μop addresses the issue of overlapping variable lifetimes. A select-μop eliminates the ambiguity of renaming by effectively postponing the renaming task. Using the select-μop ~~reduces the stall cycles while enable~~ enables renaming of registers without stalling the pipeline for disambiguating renaming, thus reducing the stall cycles. A select-μop is a single-assignment form that guarantees that every target operand is uniquely defined by only one instruction. Thus, when a variable is defined in several basic blocks throughout a control flow graph, each definition instance of the variable is subscripted to be uniquely differentiated from other definition instances of the variable. If multiple definition instances of the variable reach a common use of the variable, then a consumer instruction cannot determine which of the subscripted variables to use. For one embodiment, the compiler inserts a ϕ-node as a special placeholder at the position where two definition instances merge. The two subscripted definition variables are used as the source operands of the new ϕ-node, and a new subscripted variable is created as the new destination operand. From that point on, all subsequent uses of the variable are replaced with the new subscripted variable defined by the ϕ-node. One embodiment of subscripting and inserting a ϕ node is illustrated in Fig. 3.

Please amend the second full paragraph on page 12 as follows:

For one embodiment, the select-μop has only one destination operand, and therefore the select-μop in theory can have numerous source operands as long as the large fan-ins of the source can be efficiently implemented. For one embodiment, the select-μop has four source operands~~,~~: s0, s1, s2~~,~~ and s3. For alternative embodiments, more or less source operands could also be used. The source operands record physical register identifiers. Except for s0, each one of the source operands s1, s2, and s3 is associated with two status bits, a v-bit and a p-bit. The status bits control the selection of the source operands. The first ~~one of the~~ status bit~~s~~, the v-bit, specifies whether the register is ready. The second status bit, the ~~v-bit~~ p-bit, indicates whether the renamed definition register has been architecturally committed. The operation of the status bits is explained in more detail below.

Please amend the first full paragraph on page 13 as follows:

For an embodiment having four source operands, the processor can encounter two, three, or four instructions that define register R before generating a select-μop to resolve renaming ambiguity for register R. The generation of select-μop is triggered by two conditions. First, each one of the defining instructions, except the first defining instruction, must be guarded by unresolved predicates. And second, because the first instruction defines the default identifier, the first instruction must be ~~either~~ one of the following: An un-predicated instruction, ~~or~~ a predicated instruction whose predicate has been resolved true, or a previously generated select-μop.

Please amend the first full paragraph on page 14 as follows:

One embodiment is a method 750 of processing predicated instructions as shown in Fig. 7A. First, ~~receiving~~ a plurality of predicated instructions assigned to a common defined register in block 752 is received. At least one of the predicated instructions is out of order in a dynamic pipeline. Next, in block 754, the destination register for each one of the predicated instructions is renamed. Then, the renamed destination register with the predicate register of the predicated instruction is assigned to the source operand of a select-μop, as shown in block 756. Next, a valid predicate is determined in block 758. The register corresponding to the select-μop that corresponds to the valid predicate is selected in block 760. A consumer instruction is executed in block 762 wherein the consumer instruction uses the data from the register corresponding to the valid predicate. It will be further appreciated that the instructions represented by the blocks in Fig. 7A ~~is~~ are not required to be performed in the order illustrated, and that all the processing represented by the blocks may not be necessary to practice the invention.

Please amend the second full paragraph on page 15 as follows:

For one embodiment, the select-μops include use of a register alias table (RAT) with predicates. There are several approaches to support the generations of select-μops as described above. For one embodiment, the RAT is augmented and used in the rename stage with predicates. The RAT is used by the renaming unit to map from architectural register identifiers to physical register identifiers. When an in-flight instruction enters the rename stage, the RAT looks up the physical identifiers of the

Q10 cont'd. source operands ~~as well as~~ and assigns the result operand with a new physical identifier.

Please amend the first full paragraph on page 16 as follows:

Q11 For an alternative embodiment, a select-μop is injected only when a select-μop is required so as to avoid injecting excessive select-μops. ~~Injecting~~ In this embodiment, injection of a select-μops is demand-driven, ~~that is~~, and only occurs when more than one slot is occupied in the entry, ~~plus when either of~~ and one of the following conditions is met:

i) the use of the register is encountered at the rename stage;

ii) all slots in the entry are occupied and a new physical identifier is being allocated; or

iii) one of the guarding predicates in the slots is re-defined.

~~The use of the register is encountered at the rename stage,~~

~~Or~~

~~All slots in the entry are occupied and a new physical identifier is being allocated,~~

~~Or~~

~~One of the guarding predicates in the slots is re-defined.~~

Please amend the first full paragraph on page 19 as follows:

Q12 The outcome of the variable opt is determined by an OR operation of condition 1 and 2. However, for this embodiment, the source code was not fully rewritten for a more succinct control flow. Therefore condition 2 post-dominates condition 1, since the variable opt is assigned zero if condition 2 is true regardless of the outcome of condition

1. Even though the reverse is also true in this embodiment ~~i.e. that opt is zero if condition 1 is true despite condition 2~~ (i.e., the opt is assigned zero if condition 1 is true regardless of the outcome of condition 2), it does not necessarily translate the same in other cases. In the present embodiment the total number of cycles is 6. An embodiment more fully rewritten for more succinct control flow can further reduce the execution process to 5 cycles.

Q2 Cont'd